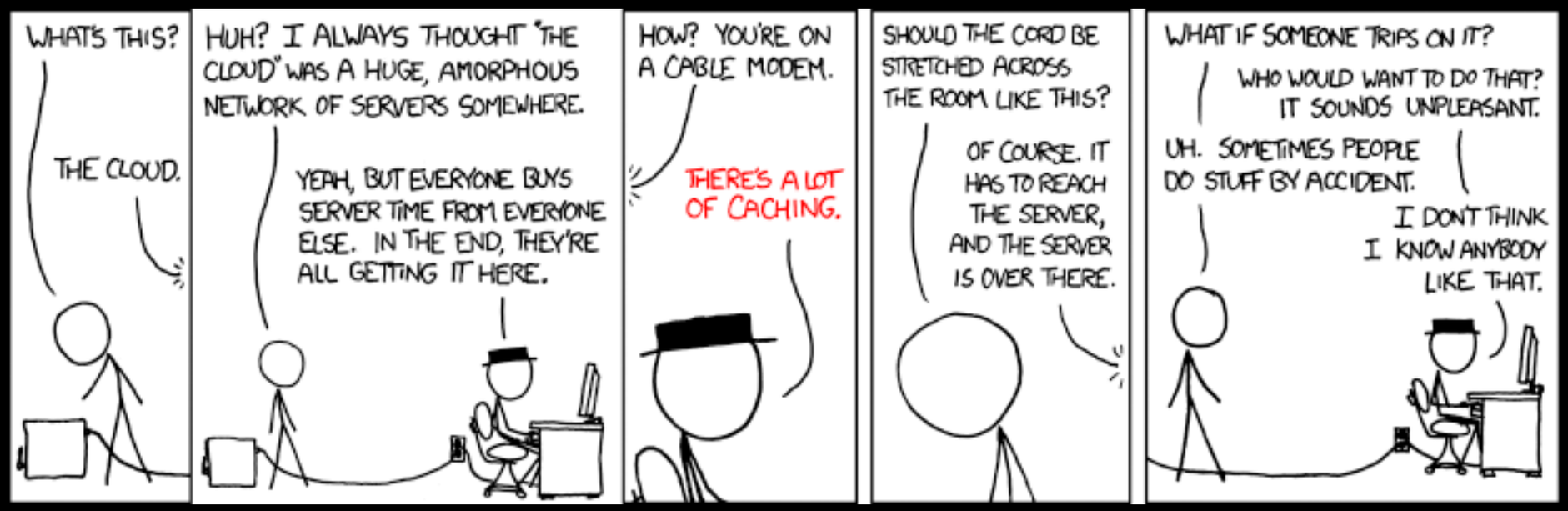
Jameson Nash
JuliaCon 2017

AoT or JIT:
How does Julia work?

Agenda

- Why infer?
- Memoization
- Incremental simplifications
- Regenerating code after adding new methods
- What is dataflow inference?

Doing things once



Many explicit levels

```
function memoize(f)
  let d = Dict()
    return function(args…)
      return get!(() -> f(args…), d, args)
    end
  end
end

const add = memoize(+)
const printonce = memoize(println)
```

Many explicit levels

```
macro memoize(f_expr)
  f_name, def = split_longdef(MacroTools.longdef(f_expr))
  d = Dict()
  return :($(esc(f_name))(args…) =
      get!(() -> ($(esc(def)))(args...), $d, args))
end
function split_function_def(ex)
  name = shift!(ex.args[1].args)
  ex.args[1].head = :tuple
  return name, ex
end

@memoize add(a, b) = a + b
```

Many internal levels

- method definition —> macros, metaprogramming
  - code_lowered —> generated functions
    simplified code structure
    - code_typed —> precompiled modules (.ji)
      global inference
      local optimization
      code_warntype – dynamic behavior annotations
      - code_llvm —> external codegen, llvmcall-2.0 Julep
        Intermediate Representation (IR) for low-level optimization
        - code_native —> static system image (.so / .dll / .dylib)
          Machine Code representation

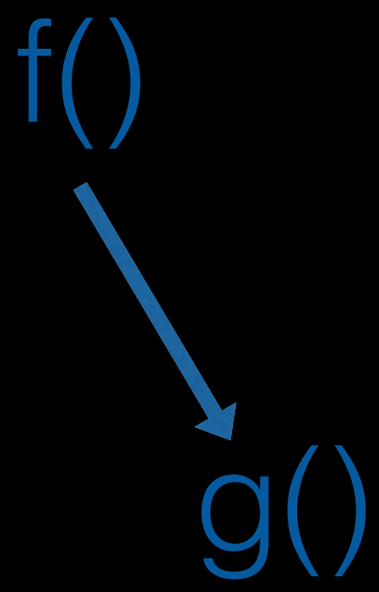
Interprocedural Optimization (IPO)

Dataflow inference in Julia: Ahead of Time optimization

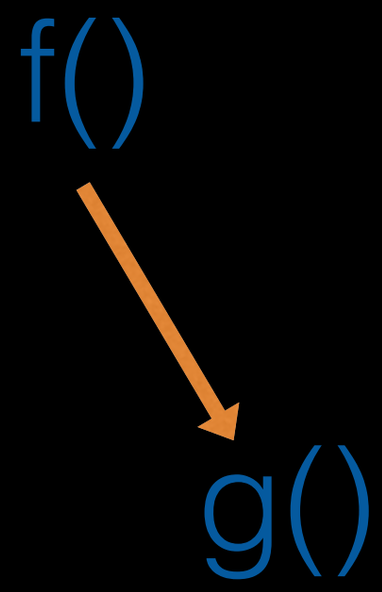
https://juliacomputing.com/blog/2016/04/04/inference-convergence.html#basic-algorithm

Edge = a call, from a caller to a callee

f() = g() ::



Backedge (“backwards edge”) = ?

f() = g() :: 

An edge lets us compute a valid return type for the code

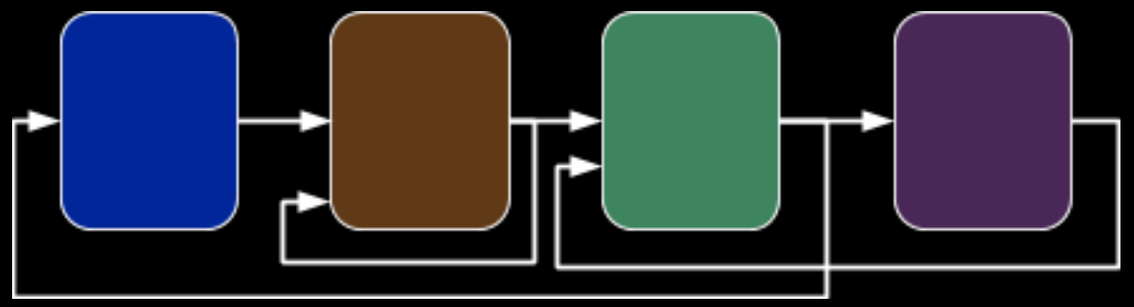
A backedge helps us compute whether the optimized code is valid

“There are only two hard things in Computer Science: cache invalidation and naming things.”

— Phil Karlton

Backedges

- Inference callers
- Inference cycles



- Inlining code / devirtualize dispatch
- Method cache (fast dispatch, compiled code)
- Method lookup (TypeMap)
- Failed method lookup (MethodError)

```
julia> const nadds = Ref(0)
Base.RefValue{Int64}(0)

julia> function count_adds()
           n1 = nadds[]
           1 + 1 + 2 + 3 + 4
           n2 = nadds[]
           return n2 - n1
       end
count_adds (generic function with 1 method)

julia> count_adds()
0

julia> function Base.:+(a::Int, b::Int)
         nadds[] = Core.Intrinsics.add_int(nadds[], 1)
         return Core.Intrinsics.add_int(a, b)
       end

julia> nadds
Base.RefValue{Int64}(821)

julia> nadds
Base.RefValue{Int64}(2680)

julia> count_adds() # JuliaLang issue 265 solved!
4
```

Method Backedges

- Adding a new method increments the world counter and starts invalidating intersecting method signatures
- Compute list of methods and old specializations that got replaced by this new method
- Remove from the method fast-dispatch caches
- Disable usage of old inference result in future worlds (truncate the max age)
- Recurse over all backedges

Signature Backedges

- Adding a new method increments the world counter and starts invalidating intersecting method signatures
- Compute all methods that were assuming this signature would cause a MethodError
- Remove from the method fast-dispatch caches
- Disable usage of old inference result in future worlds (truncate the max age)
- Recurse over all backedges

Incremental Backedges

- Precompilation adds another twist
- Can’t store backedges, invert and flatten graph, then invert again to restore
- Validate edges:
  - Check that method intersection wouldn't have returned a new method
  - ~~Recursively verify that the target result also wasn’t affected by the addition of a new method~~ (flattened)
  - Unlike typical usage, world counter on reload is not ordered. More expensive and complex world comparison required.
  - Computed by `lowerbound_dependent_world_set`: given an ordered world in the current process, determine the nearest unordered world that was visible in the compile process

Recomputing Validity

- Given a list of invalidated edges
- Decide if the method is still valid
- TODO: implement 😅

Limitations

- Currently doesn't track why invalidation occurred leading to unnecessary rework
- Hard to delete (garbage collect) old values
- Running function generators probably corrupts inference due to mis-ordering when reloading
- Incremental restoration algorithm perhaps still not entirely correct
- Doesn't require on-stack replacement

Questions?

Future Work

- Store forward edges
- Compute validation requirements on each edge:
  - Track inlining and result type separately
  - Compute minimum rework set
- Reuse native code but swap in new relocations